Symbolic Computation Using Recursive Binary Search Trees

Justin Gagnon

March 11, 2016

I define a self-similar tree-like data structure that is naturally suited to represent algebraic expressions involving the standard arithmetic operators $\{+,-,\times,\div,\hat{}\}$ in compact, factorized form. I outline an algorithm around this data structure which may be used to factorize algebraic expressions, and provide some results of an example implementation.

This article reports on a new algorithm that uses a self-similar tree structure to effectively parse and factorize algebraic expressions. A dynamically shifting mobile and computing market, coupled with increasing demands for intelligent expression parsing, calls for novel methods to efficiently and effectively parse and factorize expressions on an ever wide range of platforms. One such type of string expression is the standard algebraic math expression, involving addition ($+$), subtraction ($-$), division ($\div$), multiplication ($\times$) and exponentiation ($\hat{}$) operators, acting on c-numbers.

A common class of algebraic expressions is defined by mathematical forms composed of operators Ω acting on operands ω. Here, $\Omega = \{+,\times,\div,\hat{}\}$ – the ordinary arithmetic operators, where "$\hat{}$" represents exponentiation – and $P = \{A(O),V,\mathbb{R}\}$ may be any algebraic expression involving real numbers $\mathbb{R}$, atomic symbols V representing real-valued indeterminates, or properly-formed algebraic expressions $A(O)$ involving operands in $\{\mathbb{R},V\}$, and operators $O \subseteq \Omega$. Note that I ignore the subtraction operator "$-$" because it is trivially replaced by a multiplication operator, $-a \longrightarrow (-1)\times a$, or by an addition and multiplication, $a-b \longrightarrow a+(-1)\times b$.

First, considering expressions formed by the addition $+$ and multiplication $\times$ operators, a *sum of products* (SOP) may be defined as an expression of the type

$$A_{\text{SOP}} = \sum_i T_i(O'), \text{with } O' \subseteq \{\times, \div, \hat{}\}, \text{ and} \tag{0.1}$$

$$T_i(O') = \prod_j F_j^{(i)}(O''),\ O'' \subseteq \{\div, \hat{}\}. \tag{0.2}$$

1 Algebraic Factorization

The basic algebraic factorization problem thus consists in finding a set of common factors $\{F^{\text{c}}(O'')\}$ between a set of terms $\{T^{\text{c}}(O')\}$, and transforming the corresponding SOP form into a factored form, i.e.

$$A_{\text{f}} = \Phi\left[A_{\text{SOP}}\right] = \prod_j F_j^{\text{c}}(O'') \times \left(A_{\text{SOP}}^{(1)}\right) + \sum_i T_i^{\text{x}}(O'), \tag{1.1}$$

$$\text{where } A_{\text{SOP}}^{(1)} = \sum_i t_i^{\text{c}}(O'),\ t_i^{\text{c}}(O') = \frac{T_i^{\text{c}}(O')}{\prod_j F_j^{\text{c}}(O'')}. \tag{1.2}$$

In (1.1), the set $\{T^{\text{x}}(O')\}$ is SOP that contains no terms with common factors. The procedure defined by (1.1) is then recursively reapplied to $A_{\text{SOP}}^{(1)}$, yielding a factored form $A_{\text{f}}^{(1)} = \Phi\left[A_{\text{SOP}}^{(1)}\right]$, and so on and so forth (recursively) until it reaches a SOP $A_{\text{SOP}}^{(n)}$ that no longer contains any terms with factors in common.

One obvious problem here is that two mathematically equal expressions, e.g. $(1/a)^{-c} = a^c$, are not necessarily syntactically identical. This poses a real challenge because such expressions could potentially not be recognized as equivalent factors during the application of the factorization procedure outlined above. They are not factorizable SOPs themselves, and would therefore have to be parsed recursively in a different manner. Note furthermore that some level of lexical ordering is required, such that $a + b$ must be recognized as equivalent to $b + a$. Thus, solving the algebraic factorization problem calls for an intrinsically recursive algorithm that also makes use of strong lexical ordering.

In order to implement such a procedure, I first establish a recursive tree-like data structure – a *recursive binary search tree* (rBST). First, I will describe the rBST and illustrate some of its properties with a few examples, and then I will detail an algebraic factorization algorithm which makes use of the rBST's key properties.

2 The Recursive Binary Search Tree

Since algebraic factorization requires proper lexical ordering, it makes sense to base a solution algorithm on a binary search tree (BST) structure since a BST provides an inherently sorted representation of a data set. Moreover, since factorization is a highly recursive problem, it makes sense to employ a highly recursive data structure to solve this problem.

A *recursive* binary search tree (rBST) is a BST whose nodes, key-value pairs $N = (K_N, V_N)$, contain values which are themselves rBSTs, as in Fig. 2.1; an rBST is a tree composed of self-similar trees. For each node N, we may call its value V_N the *inner* rBST while the *outer* rBST is the tree that contains the node N.

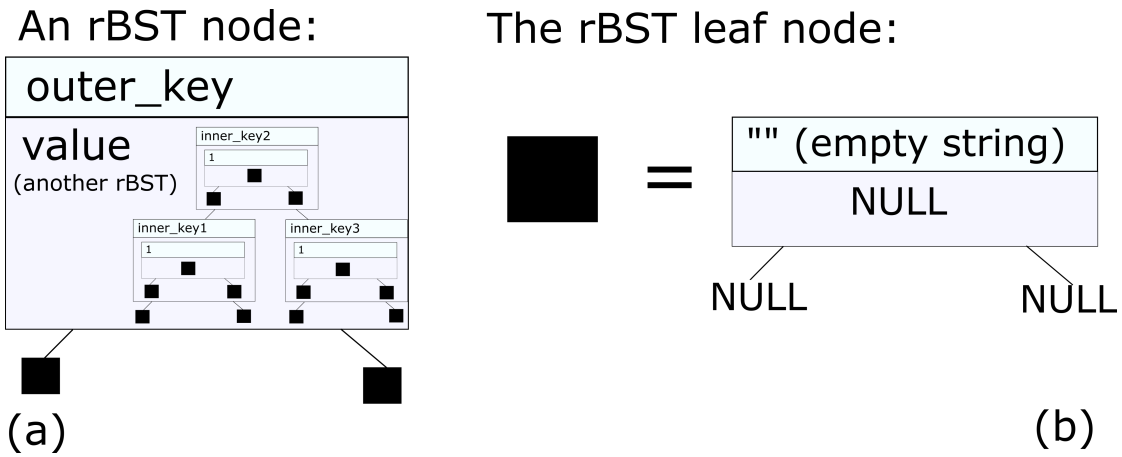

Figure 2.1: The general rBST node definition. (a) A node is composed of a key-value pair. The value is another rBST, thus making it a self-similar data structure; (b) Self-similarity stops at the leaves, which are displayed on the right, ensuring an endpoint for recursive operations performed on the tree.

Of course, as a general rule, recursion requires a base case. As such, we may define a very basic rBST: the *null* (Fig.2.1-b) which is composed simply of a root node containing an empty key string associated to a NULL inner tree, and whose left and right children are both NULL. As such, a null rBST may be considered as an empty tree.

3 The rBST as Applied to Algebraic Expressions

Given a general algebraic expression $A(O)$ (with $O = \Omega$, the full set of operators), I propose to use the rBST in the following way: The terms of an expression are the nodes of a given tree – specifically, the terms are linked together by the traditional BST structure, i.e. the path from one term to another is defined by ordinary tree traversal without entering the inner trees of any node. Thus, algebraic addition is represented as the relationship between nodes of the same tree. On the other hand, the multiplication operation is represented as the relationship between a node's key and its value (its inner rBST). In order to illustrate this structure, I provide several examples shown in Fig. 3.1. Note that numerical values are distinguished from indeterminate values by placing them at the endpoints of the rBST, right before the leaves. Since numerical values may be added and multiplied to produce new values, this tactic avoids the generation of unnecessary expressions of the form $6\times 8(2+3+a)$ upon compiling the rBST into a factored expression, as we will see later.

3.1 Examples of Expressions with their rBST Representations

Thus the rBST represents an expression by qualitatively distinguishing between two operations: the addition of terms of a given SOP, and the multiplication operation between an

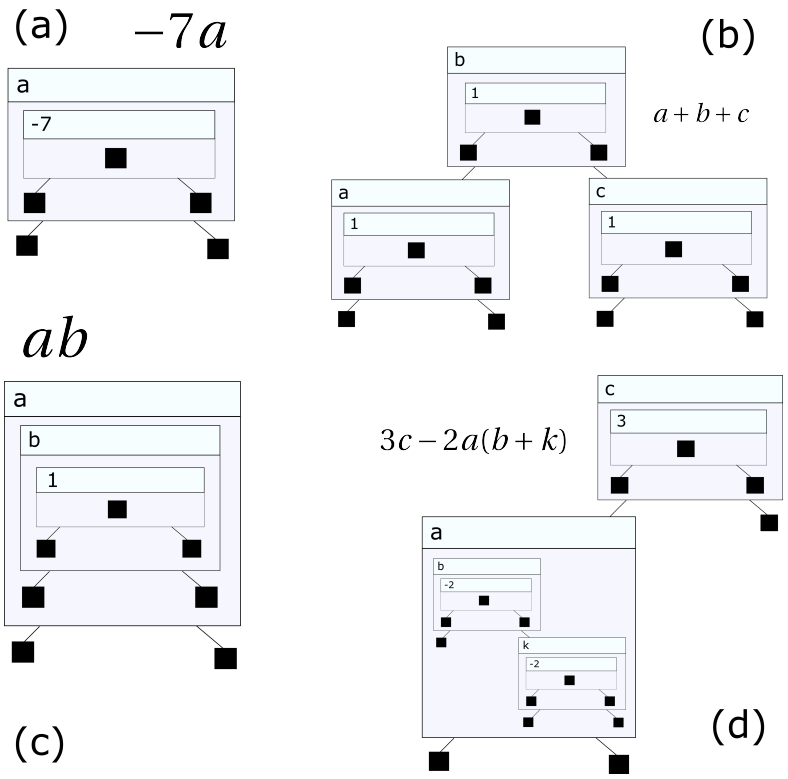

Figure 3.1: A few example trees corresponding to simple algebraic expressions: (a) $-7a$, (b) $a+b+c$, (c) ab, (d) $3c-2a(b+k)$.

atomic expression and another SOP (set of terms). Terms in the same level of a SOP are inserted as nodes of the same tree, which ensures lexical ordering between the terms. However, the above structure isn't sufficient to completely solve the factorization problem. For instance, under present considerations, a term like a^2 is not differentiated from a^3/a, or even from $a \times a$ for that matter.

The rBST representation outlined above is useful for distributed normal forms with respect to the "+" and "×" operators, but the terms themselves must be properly parsed into consistently ordered sets of factors. Fortunately, the rBST may also be used to treat sets of factors of a given term.

4 Parsing a Sequence of Factors

A single term is composed of a set of factors $A(O'')$ which are atomic, or composed of exponentiation and division operations, i.e. $O'' \subseteq \{\hat{}, \div\}$. Since factors of a given term do not appear together in the same BST, the treatment above does not ensure the lexical ordering of factors, only that of terms. To ensure that factors are consistently parsed and sorted, I also make use of the rBST.

A set of factors is treated here in a fashion similar to the above treatment of terms of a SOP. Just as the subtraction ("-") operator was previously disregarded in favor of multiplication and addition operations, the "÷" operator may be replaced by the exponentiation operator by considering that it is just an exponentiation to the power -1, i.e. $1/b = b^{-1}$ so that e.g. $a/b = a \times b^{-1}$. In this manner, the algebraic treatment of a sequence of factors may be done analogously to that of a sequence of terms in a SOP, by taking the multiplication ("×") operator as the union operator and the exponentiation ("^") operator as the intersection operator. For instance, consider the similarity between these two transformations:

$$c \times (a+b) \times d = c \times a \times d + c \times b \times d \tag{4.1}$$

$$\left(c^{a+b}\right)^{d} = c^{a \times d} \times c^{b \times d}. \tag{4.2}$$

There is, however, a key difference between the treatment of SOPs with respect to $(\times, \wedge)$ and those with respect to $(+, \times)$, and that is the non-commutativity of exponentiation of a base: $b^a \neq a^b$. This greatly simplifies the algebraic parsing of a set of factors, as one needs only to regroup the factors with respect to their bases, as in $a^b a^c / a = a^b a^c a^{-1} = a^{b+c-1}$.

4.1 Representing a Set of Factors

For a sequence of factors of the form $b_1^{a_1} \times b_2^{a^2} \times b_3^{a_3} \ldots \times b_n^{a_n}$, where (b_i, a_i) is a base-exponent pair, I implement an rBST as a "factor tree" in which each node contains a key that represents a factor's base b and a node value representing the exponent of that base, as illustrated in Fig. 4.1. Although this tree structure is lexically ordered with respect to the bases of each factor, as we will see later, it becomes ordered with respect to the exponents upon regrouping of the factors. Since the exponents are all just ordinary algebraic expressions involving all considered operators, the node values of the factor tree are just the usual rBSTs as previously defined for SOP expressions with respect to operators $\{+, \times\}$, wherein nodes are the terms and inner trees are multiplicative factors.

5 The Algebra Factorization Algorithm

In this section, I define a procedure that demonstrates how the rBST is ideally suited to produce factorized algebraic expressions. In essence, assuming a general algebraic expression $A(\Omega)$, the first step of the algorithm amounts to decomposing $A(\Omega)$ into the smallest units and throwing these pieces into the rBST, which acts as a "smart bag" as it orders the incoming pieces in a lexically consistent and algebraically correct fashion, where addition and multiplication are explicitly represented in the rBST topology.

In our original considerations, we assumed an input which is a SOP with respect to $\{+, \times\}$. Therefore, an input expression like $(a+b)(c+d)$ must first be expanded into a sum-of-products (SOP) $ac + ad + bc + bd$, where each factor is atomic with respect to $\{+, \times, \div\}$, defining an initial form for the expression. This expansion is straightforwardly performed using a recursive procedure. Following this expansion, the individual terms of the SOP are first vetted to ensure a consistent representation as factor sequences. In other words, this entails inserting

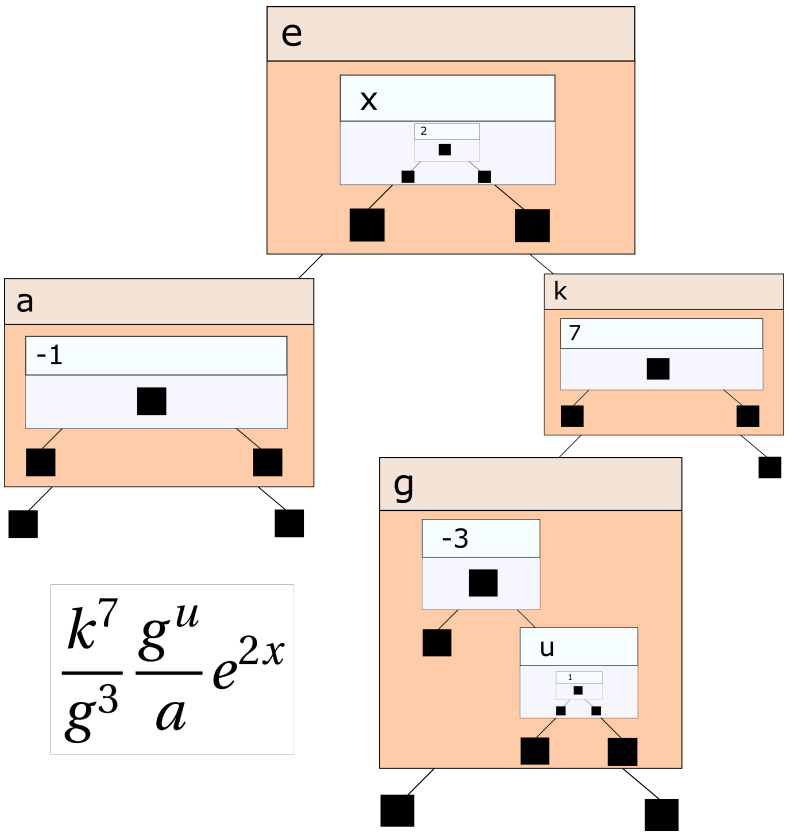

Figure 4.1: The rBST may also be used to represent a set of factors making up a single term. The keys of the outermost tree contain the bases of each factor while the values are the exponents, represented as just the ordinary additive rBSTs. The above example is an rBST representation of the expression $k^7 g^u e^{2x} / \left(a g^3\right)$.

them into a multiplicative rBST as illustrated in Fig. 4.1 (where node keys are bases and node values are exponents) and reconstructing them accordingly[1].

5.1 Parsing the Factors of Individual Terms

Since the terms of an SOP are not simply products of numbers and variable names, but rather may contain factors that encompass ÷ and ^ operators, this part of the algorithm is required in order to account for the non-trivial arithmetic of the exponentiation and division operators.

Since the multiplicative rBST uses the bases as keys in the initial tree, insertion of factors must be preceded by proper decomposition into base-exponent pairs. To insert a new base-exponent pair into the multiplicative rBST, first a search is done to find out if that base already exists in the tree. If so, the new exponent is simply added as a new term to the existing exponent (first expanded as a SOP, and then added piece-by-piece, processed recursively according to the presently discussed procedure). If the base doesn't exist in the tree, then a new node is inserted in the tree, with its key being the new base (after processing by the current algorithm) and the value being a new rBST obtained from the new exponent.

Once all factors have been inserted into the multiplicative rBST, it is compiled recursively. First, the exponential rBSTs are recompiled as additive rBSTs (using the recursive algorithm that will be discussed in the following section). Once this is accomplished, it could happen that different bases are under the same exponent (up to a sign). Thus, after compiling the bases and factors, an interim "exponent-base" tree is constructed, whose keys are now the exponents and its values are the corresponding bases from the previous multiplicative tree. This interim exponent-base tree is composed by performing an in-order tree traversal of the previously recompiled factor tree, and adding the exponents as keys to the exponent-base tree. If an exponent from the recompiled factor tree already exists in the exponent-base tree, then its base (i.e. its value in the exponent-base tree) is *multiplied* by the new base coming from the multiplicative tree.

Once the exponent-base tree has been sorted out, it is a simple matter to then recompose the factor sequence back into a single consistent unit with well-defined lexical ordering by performing an in-order tree traversal, exponentiating the node bases and node keys, and multiplying the results to achieve a single term.

5.2 Parsing and Inserting Terms of a SOP Expression into a additive rBST

Once each term of the {+, ×} SOP has been processed using the multiplicative rBST, it is now ready for proper factorization using the additive rBST. This task is performed by first initializing a additive rBST, seeding it with one of the terms in the SOP (a set of processed factors, each of them being processed using the multiplicative rBST as explained in the previous section). The subsequent terms must be added carefully. Assuming each term is decomposed into factors, a check is made to see if there are common factors between the new term and the keys in the rBST. If there is a match, then the incoming set of factors is reduced by the

[1] This of course involves use of the presently discussed algorithm to factorize bases and exponents.

common one, and the resulting insertion is then transferred to the inner tree. In symbolic form, this implies

$$\prod_i C_i \prod_i X_i + \prod_i C_i \prod_i X_i^{\text{tree}} \xrightarrow{\textit{rBST insertion}} \prod_i C_i \left(\prod_i X_i + \prod_i X_i^{\text{tree}} \right), \tag{5.1}$$

where the factors C_i represent those found to be in common between the incoming term and a tree node, and the X_i and X_i^{tree} factors are exclusive to the incoming term and the tree node, respectively.

If no match exists between the set of incoming factors and the outermost tree's key set, that is not a sufficient condition to insert a new node into the outer tree because there could exist a monomial term – a single product of several atomic factors – in the rBST's set of terms, and the additive rBST does not prescribe any lexical ordering between factors. As a monomial, any one of its factors may be equal to one of the incoming factors, and so in addition to an ordinary tree traversal, a recursive scan must be performed on those monomial terms of the tree, from the outermost factor down to the innermost tree of the monomial. If a monomial term is found, which contains a factor in common with the incoming set of factors, then a special treatment is performed: The outermost key of that monomial is swapped with the key of the common factor, and the reduced factor set (the original factor set without the matching factor) is added as a new term to the inner tree of the matching factor (which is no longer a monomial term).

5.3 The Recursive Compilation of the additive rBST

After inserting each of the terms, factor-by-factor, the final step of the algorithm involves recompiling the additive rBST into a final factorized algebraic form. For this step, we distinguish between two types of rBSTs: monomials and polynomials. A monomial is represented by an rBST which has a tree size of one, all the way to its innermost rBST. It is thus just a sequence of factors, and may be recompiled by simply multiplying its rBSTs together.

For the more general polynomial tree – for which *at least one* of its rBSTs contain more than one node – the treatment is a more complex. A general approach is to perform an in-order tree traversal and determine if the current node is a monomial term, in which case it is recompiled straightforwardly into a factor sequence. If the node is not a monomial term, such that one of its inner rBSTs contains more than one node, then the compilation is called recursively on the node's value, i.e. its inner rBST. The compiled inner tree is then eventually multiplied by the node key, and

Compilation of the rBST may be performed recursively until the key of the outermost rBST is multiplied by the compiled inner key yielding a final form. However, this treatment should be further refined because it does not automatically gauarantee that the final rBST compilation (the compilation of the rBST whose nodes all contain compiled inner trees) will not automatically guarantee an ideal factorization. Consider, for example, a tree representing the expression $a(b+c)+b+c+d$. A compilation of this tree via straight addition of its key-value pairs will generally yield a final form that is not optimally factorized, in this case because the $(b+c)$ factor appears both in the inner tree of a as well as separate terms b and c in the outer tree. Thus, a check must be made to *find any terms of the tree which contain inner trees which are*

subsets of the outer tree. This may be accomplished by performing a tree traversal and identifying such a term by comparing its inner rBST terms to the terms in the outer tree. If a term $T = [K_T, V_T]$ is found such that all the nodes of its inner tree V_T are contained in the outer tree, it means that extra refactoring may be done to simplify the precompiled rBST, in this case by simply deleting those outer nodes which overlap with those in V_T and incrementing the key K_T by 1. This amounts to a transformation such as $a(b+c)+c+d+b \longrightarrow (1+a)(b+c)+d$.
Another important edge case to consider prior to fully compiling the rBST is the following: Several terms in the tree may possess equal values, such as a tree represented by an expressions $a(b+c)+c+d(c+b)+b$. In this example, if the keys are a and d, then the factor $b+c$ is located in the inner rBSTs of keys a and d. To factorize this properly, a procedure must be implemented to precompile the tree by grouping the terms having the same values. Again, this may be accomplished with a tree traversal to find a set $\{[K_i, V]\}$ of terms with common values. If such a set is found, the value V may be fully compiled as a unit, using our current procedure, and taken as a common key K' between this set of terms, while their keys $\{K_i\}$ are inserted into a new inner tree V' and the new term $[K', V']$ is then reinserted back into the original rBST. In the example given here, this amounts to the transformation $a(b+c)+c+d(c+b)+b \longrightarrow (a+d)(b+c)+b+c$.
Note, however, that the factorization is not complete, since we end up with an extra set of loose terms $b+c$ which should be regrouped with $(a+d)(b+c)$ to yield an optimally factored expression such as $(a+d)(1+b+c)$. There are two immediate solutions to this:

1. Taking extra care in the recompilation procedure, in other words making the recompilation "smarter" by scanning a node's adjacent nodes and inner trees for common factors;

2. Re-applying the algorithm to a previously factorized unit

Option 1. is impractical due to its inherent complexity in considering various special cases and rearranging the rBST accordingly, which not only adds to the implementation complexity but also results in a greater computational cost.
Although option 2. appears to double the factorization work, it need not be the case. The first factorization step – which decomposes the original expression into a formal SOP, performs algebra on the individual terms using like multiplicative rBSTs and then inserts these into a additive rBST smart bag – handles the bulk of the lexical sorting and rearrangement of terms, such that the resulting expression is very close to being optimally factorized after the execution of the first factorization step. Thus it isn't necessary to break down the (primary) factorized expression into the smallest units (this would in fact double the overall factorization work and produce the exact same expression back again). Rather, one may instead build a simpler additive rBST by just blindly (i.e. non-recursively) decomposing the factored expression into an elementary sum of products which are not necessarily atomic with respect to $(+, \times)$, and inserting this set of terms and factors into a new rBST and compiling it into a second factorized form. As an example, consider the expression $A_0 = d - ab + db - a$. Upon first factorization, it is transformed into the somewhat better $A_1 = -(1+b)a + (1+b)d$, and attains its optimally factorized form $A_2 = (-a+d)(1+b)$ by factorizing A_1 a second time using this simpler procedure, which avoids breaking down its binomial factors $(1+b)$. After all,

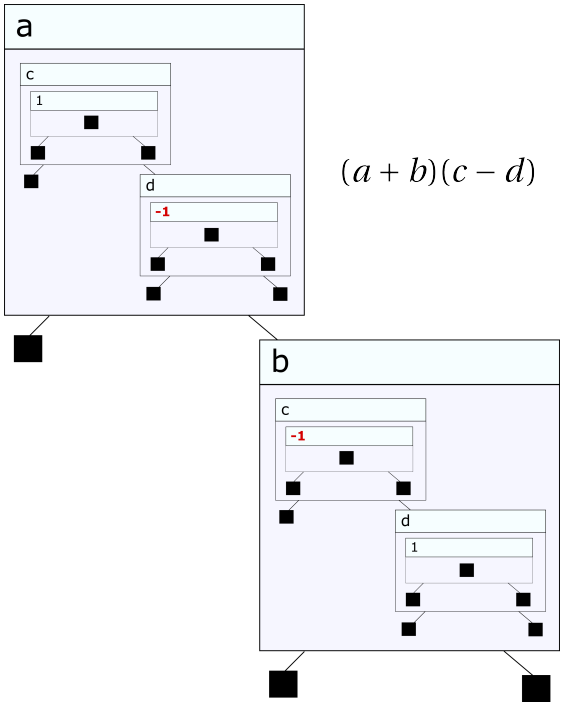

Figure 5.1: The rBST representation of $(a+b)(c-d)$. Because nodes in the tree are always additive, the -1 factors appear under different terms in the two "c,d" trees associated to a and b.

these binomials were already recomposed from the previous factorization step.

5.4 Accounting for Negative Signs in the rBST Compilation

Thus far, the negation operator has been completely discarded and replaced with the algebraically equivalent multiplication by -1. However, ignoring the -1 factors in compiling the additive rBST may prevent a complete factorization of the expression. In order to see why, consider the following expression: $(a+b)(c-d)$ which becomes decomposed to $ac-ad+bc-bd$ upon insertion into the rBST. This results in the tree displayed in Fig. 5.1. As you can see, the -1 factors are in different locations in the inner trees of a and b. Compilation of this tree will therefore produce the expression $(-c+d)b+a(c-d)$, and it stops there because the second factorization does not recognize that $-c+d=-(c-d)$.
Fortunately, this is easily corrected by ensuring, during tree compilation, that the negative sign of the first term[2] is always factorized for expressions containing multiple terms: $-a_1 \pm a_2 \pm a_3 \pm \ldots \pm a_n \longrightarrow -(a_1 \mp a_2 \mp a_3 \mp \ldots \mp a_n)$. Including this in the tree compilation, $ac-ad+bc-bd$ now becomes $a(c-d)-(c-d)b$ after the first compilation, and this ensures that

[2] Since SOPs are always sorted, their first term is well-defined with respect to lexical ordering.

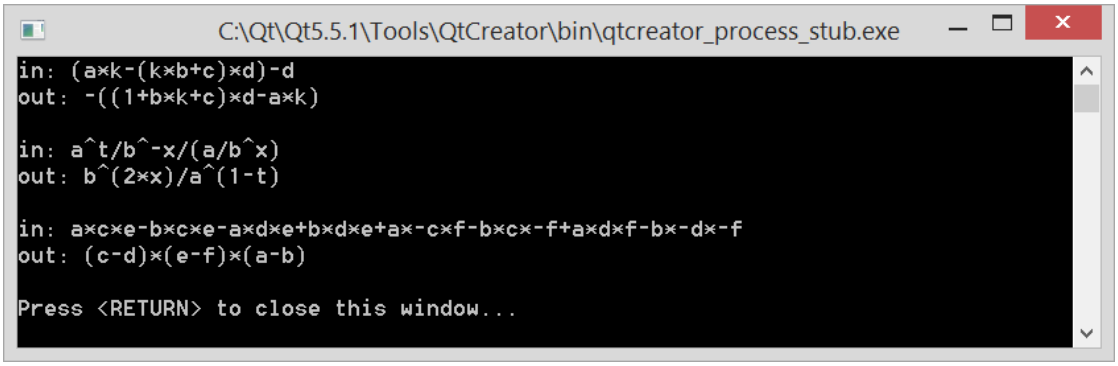

Figure 6.1: The above examples, while very simple, demonstrate the rBST algortihm's correct interpretation (and simplification) of algebra.

the second compilation (which does not fully break down the original expression into atomic pieces) now recognizes the common multiplicative factor $(c-d)$.

6 Example Implementation

I have implemented the rBST data structure and the algorithm detailed above in the C programming language. All trees are implemented as red-black trees, which are self-balancing, allowing for rapid As a first test, Fig. 6.1 represents the input and output of the algorithm for various basic test examples:
As an additional example, I take the third derivative

$$\frac{\mathrm{d}^3}{\mathrm{d}x^3}\left(\exp\left(-1/x^2\right)\sin x\right), \tag{6.1}$$

by direct application of the standard differentiation rules *without performing any algebraic simplification* while successively applying differentiation rules. This produces a long algebraic string of which occupies 18 lines of the command windows shown in Fig. 6.2. This long string was then fed to the rBST factorization algorithm which was able to simplify it down to the single-line form displayed at the bottom of the command window in Fig. 6.2, in just a fraction of a second on a modern laptop computer.

7 Conclusion

I have introduced a self-similar data structure, the rBST, which is well-suited for the interpretation and factorization of algebraic expression. By the multiplication and addition operators as the most fundamental operations in arithmetic, the rBST may be used to represent sums of terms as well as products of factors, each being amenable to algebraic simplification. In its present state, the algorithm's contraction of the products to exponentials, e.g.

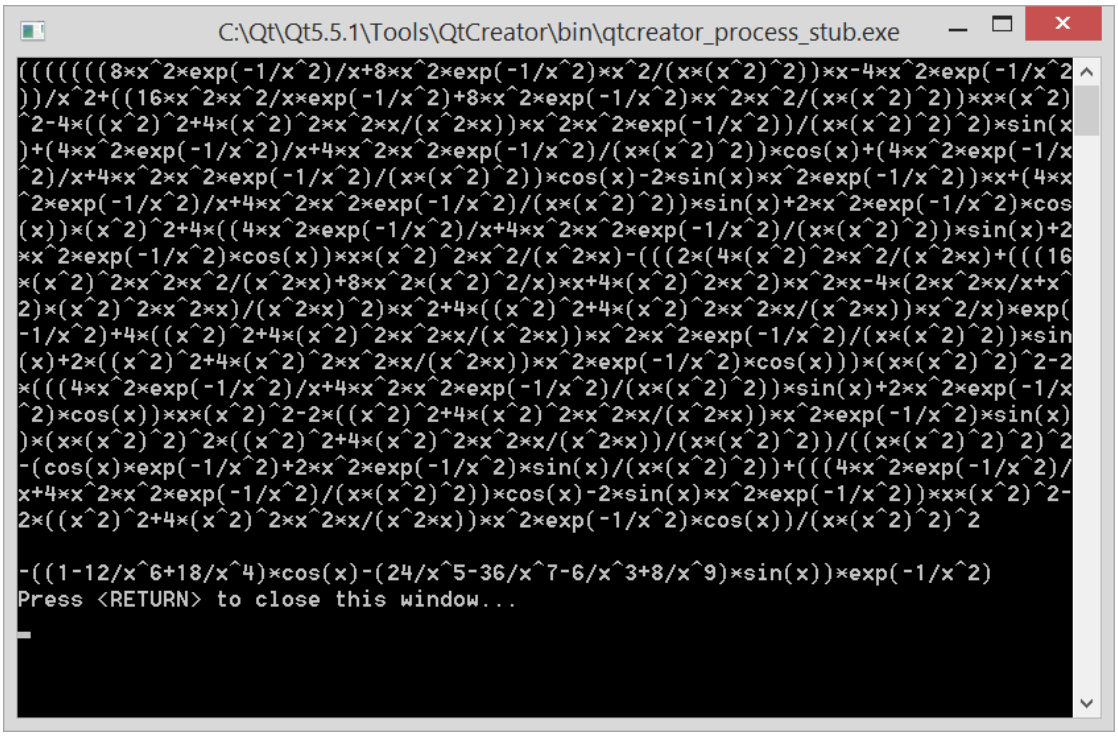

Figure 6.2: The top string is the result of direct, *unsimplified* application of differentiation rules to evaluate (6.1). This string is then given to the rBST algorithm which produces the bottom line.

$(a+b)*(a+b) \longrightarrow (a+b)^2$ is non-reversible: exponential factors are never expanded. As a result, the algorithm's factorization of complex SOPs, which may be expanded versions of *products of sums*, such as an expanded version of $(a-b)(c-d)(e-f)(g-h)$, is limited because it may contract a product of two equal terms as a squared quantity, which is irreversible. Further development will account for such edge cases. As well, since the compilation of a tree back into a factored string expression is computationally expensive, a computer program that stores expressions would ideally keep them as in-memory rBST structures, and convert them into string expressions only when needed. Thus, this may entail a modification of the rBST node structure.
More importantly, since an rBST may be used as a representation of an algebraic expression, it would call for the development of a type of arithmetic for rBSTs that treats addition, subtraction, multiplication, division, and exponentiation directly between tree structures so as to avoid converting back-and-forth between rBST and string representations.

The author gratefully acknowledges Peter Renkel for valuable discussions and insights.

This article will be later updated to include a bibliography